

CSE 509: COMPUTER SYSTEMS SECURITY

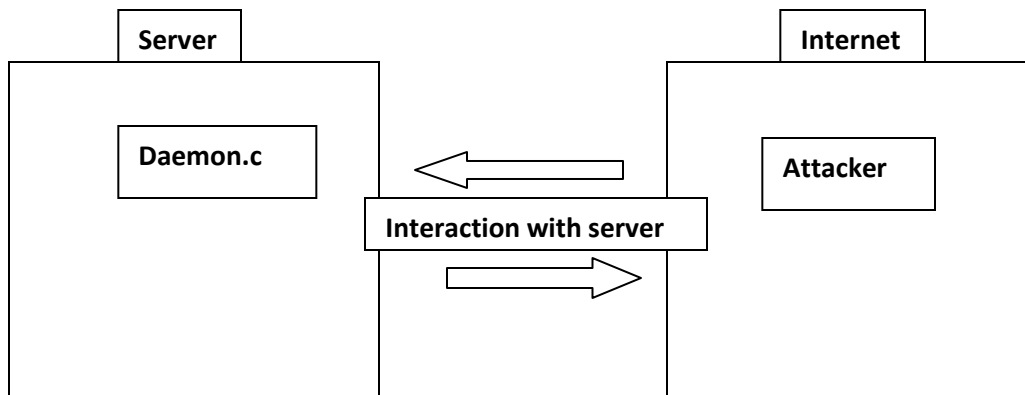
SPRING' 09: LECTURE NOTES

Lecture 7: Introduction to Software Security

Date: 3/16/2009

Buffer Overflows

1. The attacker makes the server execute the attacker's code
2. The Daemon running on the server must restrict the attacker from doing so.



Organization of Virtual Memory in a typical 32 bit Linux System

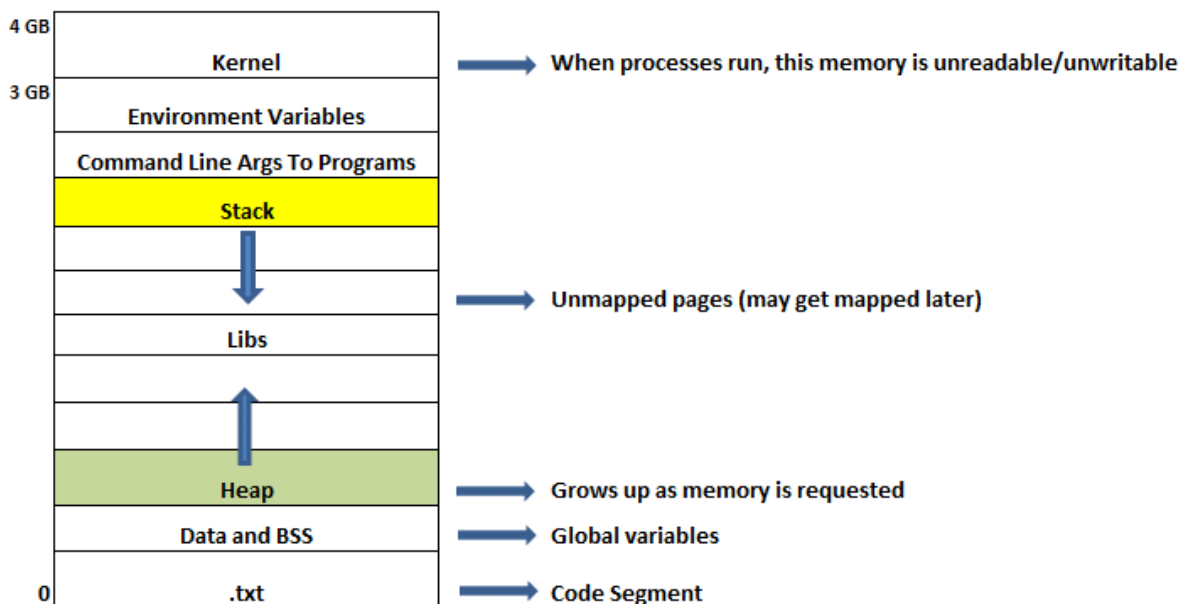


Figure 1 : Virtual Memory Layout

Memory snapshots such as above may depend on the program and the OS State. Attacker may have detailed knowledge of the memory layout if he is running the same OS and the same programs as the machine he wishes to subvert.

Zooming in on the stack

Suppose daemon runs the code:

```
void get_user (char *buf)
{
    int len;
    printf("username:");
    scanf("%s",buf);
    .
    .
    .
}
void login (void)
{
    char username[64];
    get_user (username);
    .
    .
}
```

The stack layout in this case looks like fig. 2

Every CPU has a stack pointer, SP which points to the lowest allocated address on the stack.

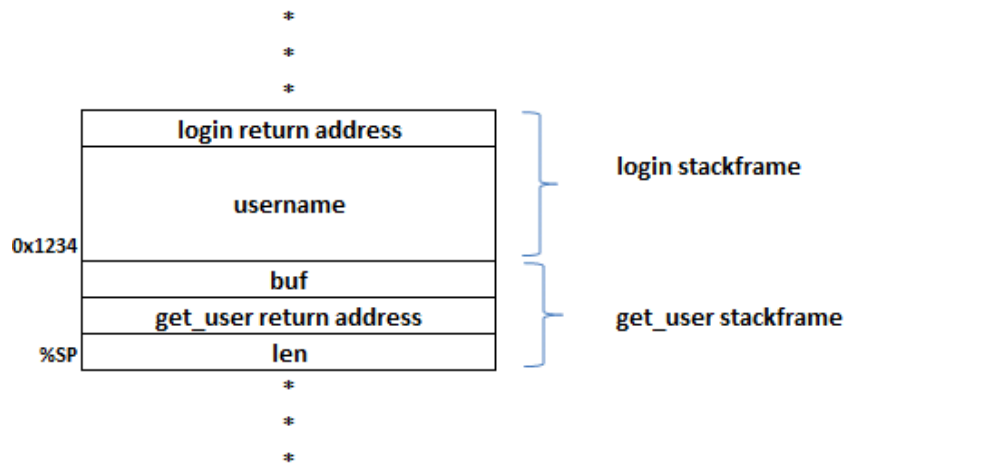


Figure 2 : stack snapshot

For the stack snapshot (fig. 2) a malicious input could be:

<machine code> <0x1234>

Note: the machine code will be 64 bytes. Basically the intent of the attacker is to plant 64 bytes of his code in username and make the control jump to his code.

Original return address will be overwritten by 0x1234 (due to the overflow of the input in login return address). Therefore if the get_user() doesn't crash and all goes well with the login() function as well, the control will be transferred to the attacker's code (starting at 0x1234).

0x1234 is the address chosen by the attacker because it has high probability of being in the 'username'.

Line of Attack → from the attacker's perspective

1. Attacker might add **Landing Pads** or NOPS before the machine code to ensure that the chances of the attack being successful increase. He might use the following input:

< 4 NOPS> <60 bytes of machine code> <0x1234>

2. Brute force → attacker might try various addresses in place of 0x1234.

3. Trampolines

Suppose register %r0 has address of the username (when login returns)

Suppose .txt segment contains instruction " jmp %r0" at 0x0122

(Normally the address of the text segment does not change often and so it is reasonable to assume that an attacker running the same code on a similar machine would be able to see the exact same .txt segment)

So the attacker simply writes 64 bytes of machine code followed by 0x0122 (attacker does not even need to know the address of username).

Therefore once login returns, it jumps to 0x0122, encounters the 'jmp' instruction and finally jumps to the address of the username.

Solution :

Making the stack RW (not executable) can prevent trampoline attacks because the machine code cannot be executed anymore.

(Refer Fig. 3 for RWX permissions)

So what does the attacker do now??? He resorts to return-to-libc attack (next attack)

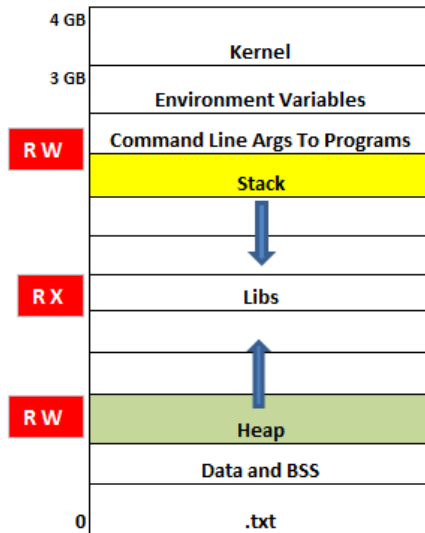


Figure 3: Avoiding Trampoline Attacks

4. return-to-libc attack

Attack philosophy:

- Attacker doesn't add his own code
- Instead injects data
- Configures return to pre-existing code
- The code that is usually executed is that of "system" library call which executes a command string in a shell.

Consider the following implementation of 'system' library call:

```
system (char *arg)
{
    exec (arg);
}
```

The attacker knows the exact address of this function and also its machine code (shown below):

```
0x1122    mov %r0 , [%SP + 4]
          int 0x80
```

Now suppose the attacker wants to wipe off the whole stack and make it look like the following:

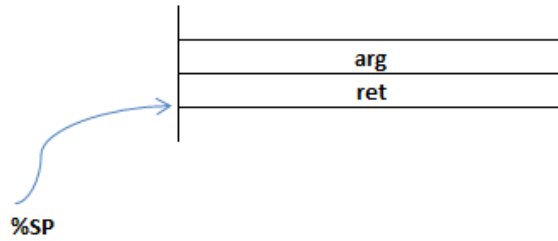


Figure 4: Attacker's view of a vulnerable stack

ret : address of the "system" library call

arg: value of the attacker's choice

Relating this view with our example, the attacker could provide the following input string:

<command> <0x1122> <dummy return address> <0x1234>

This input will result in the following stack view (quite like the one shown in fig. 4) → the attacker wins.

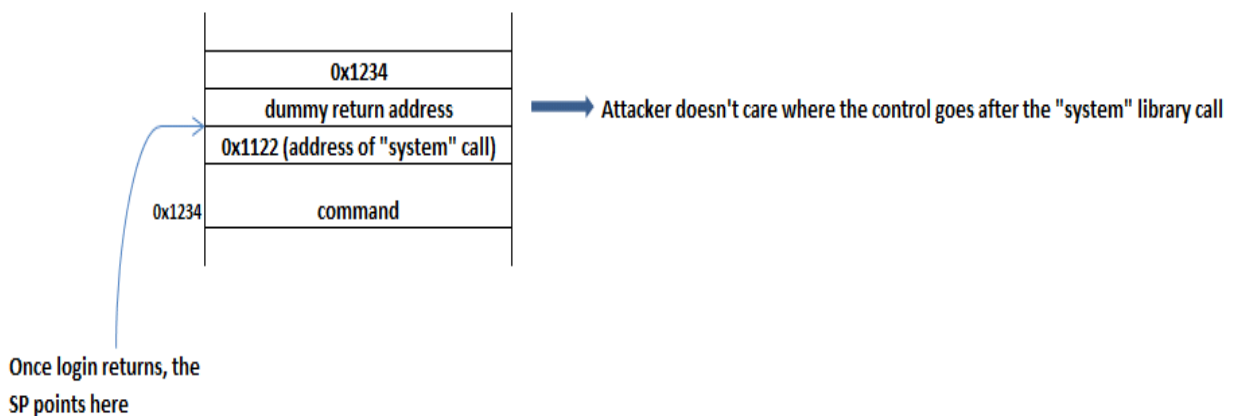


Figure 5: Stack Layout after the attacker inputs the malicious string

Fig. 5 shows how the malicious input string causes the control to jump to 0x1122 (address of "system" library call). The system call puts the value at SP+4 (which is 0x1234) in r0, causing the "system" library call to execute the command.

General techniques to make code snippets execute:

Function pointers

- Callbacks (local variables)
- Structures (full of function pointers)
- V tables / V pointers (C++)
- Execution Handlers
- Combine with return-to-libc attacks
- At exit()

Other Buffer Overflows:

→ Arbitrary Memory Write

```
void func()
{
char buf [64];
int val;
int *ptr;

<buffer overflow of buf>
*ptr = val;
}
```

Now in the above code snippet, if buf overflows , then it may spill into the value of val.

→ Heap Overflows

Similar to arbitrary memory write attacks. Heap provides the memory that is allocated through malloc.

The arrangement in a heap is as follows:

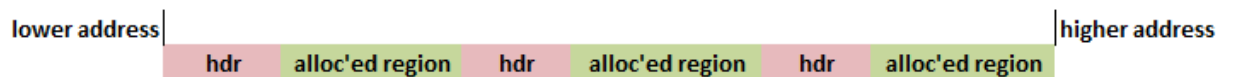


Figure 6: layout of heap allocation

The hdr structure is arranged as a doubly linked list. Now when a free operation occurs (for freeing memory), the following code is executed:

```
hdr->next->prev = hdr->prev;
```

Just like in the above attack (arbitrary memory write attack), if the attacker can overflow the alloc'ed field and modify hdr->prev appropriately, he can control the final location, where hdr->next->prev points to [analogous to val in the previous attack]